Peter Wollschlaeger

Wie man in Basic mit Binärzahlen umgeht

Basic-Interpreter sind im allgemeinen nur für den Umgang mit Dezimalzahlen ausgelegt. Häufig kommt es aber vor, daß der Zustand einzelner Bits oder das hexadezimale Äquivalent eines Bitmusters interessiert. Hier lassen sich die booleschen Operationen vorteilhaft einsetzen. Elegante Programmiervorschläge finden Sie in diesem Beitrag.

Stellt man einem Programmierer die Aufgabe, eine Dezimalzahl in eine binäre zu konvertieren und das Ergebnis als Bitmuster auszudrucken, so erinnert sich mancher sofort seiner mathematischen Grundausbildung, und es entsteht ein Programm nach dem Prinzip der fortlaufenden Division. Das Ergebnis ist eine recht lange und gerade für Basic-Interpreter zeitintensive Routine.

1. In der Praxis benötigt man nur Konversionen $\leq 2^{16}$ (Adreßbusbreite = 16).
2. Im Integer-Format wird eine Zahl direkt in zwei Bytes gespeichert.
3. Da in diesem Format im 2er-Komplement gearbeitet wird und das MSB des höherwertigen Bytes als Vorzeichen dient (gesetzt ≙ negativ), ist die größte darstellbare Zahl $2^{15}-1=32767$, entsprechend die kleinste -32768.

D. h., sofern man sich innerhalb dieser Grenzen bewegt, steht das gesuchte Bitmuster bereits im Arbeitsspeicher. Es liegt also nahe, das Muster direkt aufzuspüren.

Dies ist beim TRS-80, PET, CBM oder allgemein in Microsoft-Basic kein Problem. Hier lassen nämlich die logischen Operatoren AND und OR nicht nur Vergleiche der Form „IF A=B AND C=D THEN..." zu, sondern auch den Vergleich Bit für Bit nach der UND- bzw. ODER-Funktion. Bei Maschinen der oberen Preisklasse (z. B. TEK 4051/52) ist das eher die Ausnahme. Diese Rechner reduzieren vorab die Argumente der Operation auf 0 oder 1 (Grenze 0,5), andere bieten zusätzliche Befehle wie BINAND und BINOR.

Ob eine Maschine UND bitweise berarbeitet, ist leicht festzustellen. Geben Sie ein: PRINT 3 AND 7

Das Ergebnis sollte 3 sein, weil

```
dezimal  3 = binär 011
AND      7 = binär 111
ergibt             011 = dezimal 3
```

Die UND-Funktion an sich ist einem Eletroniker geläufig, doch nicht jeder ist Assembler-Programmierer. Deshalb folgende Erläuterung: Nehmen wir an, man verknüpft folgende Werte:

```
      0 1 1           0 0 1
  AND 0 1 0       AND 0 1 0
    = 0 1 0         = 0 0 0
```

Die jeweils erste Zeile ist das zu testende Bitmuster, die jeweils zweite nennt man Maske. In diesem Beispiel wird das zweite Bit maskiert. Danach ist das Ergebnis 0 oder 1 am zu prüfenden Bit je nachdem, ob es gesetzt war oder nicht.

```
10 INPUT"DEZIMALZAHL";D
20 FOR I=7 TO 0 STEP-1
30      PRINTSGN(D AND 2↑I);
40 NEXTI
50 PRINT:GOTO 10
```

Bild 1. Umwandlung einer Dezimalzahl ≤ 255 in ein Binärmuster

```
10 INPUT"DEZIMALZAHL";D
11 X=INT(D/256) : GOSUB 20
12 X=D-256*X    : GOSUB 20
13 PRINT:GOTO 10
20 FOR I=7 TO 0 STEP-1
30      PRINTSGN(X AND 2↑I);
40 NEXTI
50 RETURN
```

Bild 2. Umsetzung einer Dezimalzahl ≤ 65535 in ein Binärmuster

```
10 INPUT"BINAER-MUSTER";B$
20 L=LEN(B$) : D=0
30 FOR I=1 TO L
40       A=VAL(MID$(B$,I,1))
50       D=D+A*2↑(L-I)
60 NEXTI
70 PRINTD : GOTO 10
```

Bild 3. Ein Binärmuster, das als String B$ vorliegt, wird in eine Dezimalzahl umgesetzt

Numeriert man in einem Byte von 8 Bit Länge die Bits von rechts nach links, beginnend bei 0 und endend bei 7, so repräsentiert die „Nummer" als Exponent zur Basis 2 den Stellenwert. D. h. um zu prüfen, ob Bit 3 gesetzt ist, muß man nur mit $2^3=8$ UND-verknüpfen oder allgemein für Bit n mit 2^n.

Genau diesen Effekt nutzt das Basic-Programm in *Bild* 1 aus. Der richtigen Schreibweise wegen beginnt es allerdings bei Bit 7. Ist Bit 7 gesetzt, ist das Ergebnis $2^7=128$; 0 wenn es nicht gesetzt ist. Nun, wir wollen nicht 128 angezeigt bekommen, sondern 1 oder allgemein für jedes gesetzte Bit n nicht 2^n, sondern 1. Dies läßt sich in Basic leicht mit der SGN-Funktion realisieren.

Rein theoretisch ist das Verfahren für Zahlen bis $2^{15}-1=32767$ gut, man könnte also die Schleife auch bei 14 beginnen lassen. Dies funktioniert beim CBM einwandfrei, beim TRS-80 nicht. Zumindest das Gerät des Autors beginnt ab 2^{11} unkorrekte Ergebnisse zu liefern. Eingangs wurde von 2^{16} gesprochen, und das überfordert bei dieser Lösung auch den CBM. Deshalb die Lösung nach *Bild* 2. Die Dezimalzahl wird in zwei Bytes zerlegt; das Programm aus Bild 1, zum Unterprogramm ernannt, wird zweimal durchlaufen. Mit der Variablen X in Zeile 11/12 wird erst das höher- dann das niederwertige Byte übergeben.

Eine Dezimalzahl wird in zwei Bytes zerlegt

Genau genommen ist eine Dezimalzahl in zwei andere zu zerlegen, so daß die eine dem Dezimahlwert des höherwertigen, die andere dem des niederwertigen Bytes der rechnerinternen 2-Byte-Darstellung entspricht. Die Aufgabe steht immer an, wenn Basic die Startadresse eines Maschinenprogramms in den dafür vorgesehenen Sprungvektor laden muß.

Da hierbei häufig recht umständliche Wege gegangen werden, kurz die zwei einfachsten Lösungen:

1. Ist die Startadresse S ≤ 32767 und soll das niederwertige Byte nach 16526, das höherwertige nach 16527 (USR-Vektor des TRS-80), dann reicht:
   10 POKE 16526,S AND 255:POKE 16527, INT (S/256)
   Beim ersten POKE-Befehl wird mit der Maske 0000000011111111 = 255 das höherwertige Byte ausgeblendet. Die Division durch 256 im zweiten POKE-Befehl resultiert aus folgender Überlegung: Im niederwertigen Byte gelten die Stellenwerte 2^0 bis 2^7, im höherwertigen Byte 2^8 bis 2^{15}. D. h.

zwei Bits auf jeweils gleicher Stelle in diesen beiden Bytes unterscheiden sich immer durch $2^8 = 256$. Bei gleichem Inhalt ist das höherwertige Byte um den Faktor 256 größer, folglich muß der ganzzahlige Quotient der Zahl dividiert durch 256 im höherwertigen Byte stehen.

2. Ist die Startadresse $S > 32767$, scheitert AND an den Grenzen der 2er-Komplement-Darstellung. Deshalb wird zuerst das Dezimaläquivalent des höherwertigen Bytes errechnet und dann als der verbleibende Rest das niederwertigen Byte, also:
   10 H = INT(S/256) L=S−256H
   20 POKE 16526,L : POKE 16527,H

Binär/Dezimal-Umsetzung

Gemeint ist hier die Übersetzung eines Bitmusters von Nullen und Einsen in eine Dezimalzahl. Die Lösung von *Bild 3* ist einfach. Das Bitmuster steht im String B$. In der Schleife wird mit Hilfe der MID$-Funktion (in anderen Basic-Versionen SEG) bitweise aufgelöst.
Da die Substrings lediglich die ASCII-Zeichen von 0 und 1 repräsentieren, werden sie mit Hilfe der VAL-Funktion umgewandelt. Zeile 50: Die Länge des String minus Laufindex der Schleife (L-I) ist der Exponent zur Basis 2, der mit dem Stelleninhalt A multipliziert wird. Das Produkt wird jeweils zu D addiert.

Zahlen beliebiger Basis dezimal dargestellt

Die Routine in Bild 3 ist die Basis für die Universallösung in *Bild 4*. Die Value-Funktion in Zeile 40 erlaubt keine größere Basis als 9. Deshalb wurde sie durch die ASCII-Funktion (ASC) ersetzt. Sie liefert die Werte 48 bis 57 für die Ziffern 0 bis 9 und 65 bis 90 für die Großbuchstaben A bis Z. Man muß also prüfen, ob der Wert > 64 ist (Zeile 41). Ist das der Fall, ist 55 zu substrahieren, also aus dem Buchstabe A wird 65−55=10, aus B wird 11 usw. Trifft die Bedingung A > 64 nicht zu, ist es eine Ziffer, und es wird 48 subtrahiert. In Zeile 50 wurde lediglich die Konstante (Basis 2) durch die Variable B ersetzt.

Gewünscht: Hexadezimalzahlen

Meist trifft man in solchen Routinen auf den String „0123456789ABCDEF" und auf viel Mathematik. Entsprechend lang und langsam werden die Routinen. Rechnet man direkt mit ASCII-Zeichen und wendet wieder UND-Masken an, wird's einfacher und schneller.
Grundüberlegung: Die Dezimalzahl steht als 16-Bit-Muster im Arbeitsspeicher. Jeweils 4 Bit sind in die Zeichen 0 bis F umzuwandeln. Teilen wir erst einmal die 16 Bits in zwei Bytes (Zerlegen in zwei Bytes wie beschrieben) und dann jedes Byte in zwei Halbbytes.
Das niederwertige Halbbyte erhält man, indem man das Byte mit der Maske 00001111 UND-verknüpft also in Basic: AND 15 (siehe *Bild* 5, Zeile 100). Mit dem höherwertigen Halbbyte ist es nicht ganz so einfach. Es wird auch maskiert, nämlich mit 11110000 = 240, nur steht es danach noch 4 Bit zu weit links, also in falscher Wertigkeit. In Assembler würde man jetzt mit vier Shift-Befehlen das ganze nach rechts rücken, jedoch in Basic sind Interpreter/Compiler mit einem Shift-Befehl eher die Ausnahme, aber dividieren können sie alle. Erinnern wir uns, daß ein Shift nach rechts einer Division durch 2 entspricht, so gilt für 4: Dividiere durch 16, womit Zeile 100 klar ist.
Die folgenden 4 Zeilen wandeln nur noch die numerischen Werte in ASCII-Werte um, wobei die Technik aus dem Programm in Bild 4 hier umgekehrt wird (wer einen Computer besitzt, der ELSE versteht, kann zwei Zeilen sparen). In Zeile 150 werden mit Hilfe der CHR$-Funktion aus den ASCII-Werten die letztlich zu druckenden Zeichen.

Noch etwas Assembler

Wie kaum noch zu verheimlichen, ist der Autor von Assembler zu Basic gekommen. Deshalb sei in *Bild* 6 die Routine gezeigt, die den Anstoß zu den Basic-Programmen gab. Sie ist in Z80-Assembler geschrieben aber fast 8080-kompatibel (JR durch JMP ersetzen). Der Autor nutzt die Routine, um während der Programmausführung Speicher/Registerinhalte zu testen. Nach jedem „CALL HEX" wird der Inhalt von HL in Hex.-Darstellung auf dem Schirm angezeigt.
In Basic würde man die Zeilen 20–26 so schreiben:
20 A = A AND 15
21 IF A < 10 THEN 25
22 :
23 A = A+55
24 GOTO 26
25 A = A+48
26 GOSUB ...
Wer sagt da noch, daß Assembler schwierig sei?

```
10 INPUT"ZAHL";B$
11 INPUT"BASIS";B
20 L=LEN(B$) : D=0
30 FOR I=1 TO L
40   A=ASC(MID$(B$,I,1))
41   IF A ) 64 THEN A=A-55 : GOTO 50
42   A=A-48
50   D=D+A*B+(L-I)
60 NEXTI
70 PRINTD : GOTO 10
```

Bild 4. Umsetzung einer Zahl beliebiger Basis in eine Dezimalzahl

```
10 INPUT"DEZIMALZAHL";D
20 B=INT(D/256) : GOSUB 100
30 PRINT H$;
40 B=D-B*256     : GOSUB 100
50 PRINT H$ : GOTO 10
100 L=B AND 15 : H= (B AND 240)/16
110 IF H)9 THEN H=H+55 : GOTO 130
120 H=H+48
130 IF L)9 THEN L=L+55 : GOTO 150
140 L=L+48
150 H$=CHR$(H)+CHR$(L) : RETURN
```

Bild 5. Dezimal/Hexadezimal-Umsetzung

```
                 00001 ;UNTERPROGAMM HEX
                 00002 ;INHALT VON HL WIRD AUF DEM SCHIRM ANGEZEIGT
                 00003
0033             00004 PRINT   EQU     33H              ;UP ANZEIGE BYTE IN AKKU
7FBC             00005         ORG     7FBCH
7FBC 7C          00006 HEX     LD      A,H              ;HIGHER BYTE MIT
7FBD CDCD7F      00007         CALL    SHIFT            ;   HIGHER NIBBLE
7FC0 7C          00008         LD      A,H              ;NOCH'MAL HIGHER BYTE MIT
7FC1 CDD17F      00009         CALL    ANZ              ;   LOWER NIBBLE
7FC4 7D          00010         LD      A,L              ;LOWER BYTE MIT
7FC5 CDCD7F      00011         CALL    SHIFT            ;   HIGHER NIBBLE
7FC8 7D          00012         LD      A,L              ;NOCH'MAL LOWER BYTE
7FC9 CDD17F      00013         CALL    ANZ              ;   MIT LOWER NIBBLE
7FCC C9          00014         RET
                 00015
7FCD 1F          00016 SHIFT   RRA                      ;SCHIEBE
7FCE 1F          00017         RRA                      ;  HIGHER NIBBLE
7FCF 1F          00018         RRA                      ;     4 BIT NACH
7FD0 1F          00019         RRA                      ;         RECHTS
7FD1 E60F        00020 ANZ     AND     15               ;HIGHER NIBBLE WIRD 0
7FD3 FE0A        00021         CP      10               ;WERT ) 10 ?
7FD5 3804        00022         JR      C,ZIF            ;NEIN: ZIFFER
7FD7 C637        00023         ADD     A,55             ;BUCHSTABE
7FD9 1802        00024         JR      DISP             ;ZUR ANZEIGE
7FDB C630        00025 ZIF     ADD     A,48             ;ZIFFER
7FDD CD3300      00026 DISP    CALL    PRINT            ;ANZEIGE 1 CHR IN AKKU
7FE0 C9          00027         RET
0000             00028         END
00000 TOTAL ERRORS
DISP     7FDD
ZIF      7FDB
ANZ      7FD1
SHIFT    7FCD
HEX      7FBC
PRINT    0033
```

Bild 6. Z80-Maschinenprogramm: Anzeige eines Registerinhalts in Hexadezimaldarstellung

Hans-Georg Joepgen

String-Mathematik vertausendfacht Rechengenauigkeit

Handelsübliche Mikrocomputer und die meisten Programmiersprachen erlauben nur Berechnungen mit höchstens rund einem Dutzend signifikanter Stellen im Ergebnis. Was danach kommt, wird gerundet oder, schlimmer, weggelassen. Der folgende Beitrag stellt ein alternatives Rechenverfahren vor, das es erlaubt, die Anzahl signifikanter Digits fast beliebig zu erhöhen.

Die heutigen Basic-Interpreter besitzen zumeist eine recht begrenzte Rechengenauigkeit: In DAI-Basic, im Focal-Dialekt FCL und in gewissen Pascal-Systemen ist nach sechs signifikanten Stellen Schluß, Microsoft-Basic-Dialekte tun's je nach Version mit acht bis sechzehn Ergebnis-Digits. *Bild 1* zeigt, wie sich beispielsweise eine in Südwestdeutschland als Schulcomputer verbreitete Maschine benimmt, soll sie zu großen Zahlen eine Eins addieren: Die unter dem Microsoft-Basic-Dialekt PALSOFT laufende ITT-2020 unterschlägt in der Ausgabe solange den Zusatzeinser, bis die Testzahl A die Milliardengrenze nach unten überschritten hat. Für gewisse Aufgabenstellungen beispielsweise aus der Molekularbiologie, wo die statistische Behandlung kleiner Veränderungen großer Zahlen von Bedeutung ist, sind derlei rüde Verkürzungen der Wahrheit durch den Computer schlichtweg untragbar. Verbreitete Methoden der Abhilfe in solchen Fällen sind das Ausweichen auf Assembler-Programmierung oder unhandliche Integer-Arrays, die allenfalls von einigen wenigen Spezialisten geübt gehandhabt werden – beides mit mancherlei Nachteilen behaftet. Gesucht wurde deshalb nach einem Verfahren, das die Erhöhung der Rechengenauigkeit unmittelbar in Basic bringen sollte, und zwar ohne Unterschied in so divergierenden Dialekten wie den bei den Maschinen DAI, TRS-80, PET, Apple, ITT-2020 und TI-99/4 implementierten Versionen. Die Wahl fiel auf eine Art der Ziffernverarbeitung über Zeichenketten, wie sie als „String-Mathematik" hier vorgestellt wird.

```
1  REM GENAUIGKEITSTEST
2  REM ------------------
3  REM
4 A = 112233445566: PRINT  CHR$ (4)"PR#5
": LIST : PRINT " ": PRINT " ": PRINT "
": PRINT " TESTZAHL A ","SUMME A+1": PRI
NT "------------------------------"
5  FOR N = 1 TO 4: GOSUB 6: NEXT : PRINT
 "==============================": PRIN
T  CHR$ (4)"PR#0": END
6  PRINT A,A + 1:A = A / 10: RETURN


 TESTZAHL A      SUMME A+1
------------------------------
1.12233446E+11   1.12233446E+11
1.12233446E+10   1.12233446E+10
1.12233446E+09   1.12233446E+09
112233446        112233447
==============================
```

Bild 1. Ein Versuch beweist: Bei Zahlen von einer Milliarde an aufwärts kommen Computer ins Lügen und unterschlagen ohne Warnung Stellen

Die Lösung: Fritzchens Schulheft-Algorithmus

String-Mathematik bedeutet: Zahlen werden nicht länger rechnerintern im üblichen Binär-Format als Mantisse und Exponent gespeichert (dies führt zum Grundübel der fehlenden Stellen), sondern als ASCII-Zeichenkette. Da die genannten Rechner das unmittelbare La-

```
]
]
1  REM VERFAHRENSPRINZIP
2  REM ------------------
3 S1$ = "112233445566778899"
4 S2$ = "192939495969798999"
5 S$ = "":SS$ = "0"
6  FOR N = 18 TO 1 STEP  - 1
7 S% =  VAL ( MID$ (S1$,N,1)) +  VAL
     ( MID$ (S2$,N,1)) +  VAL ( LEFT$
     (SS$,1))
8  IF S% > 9 THEN S% = S% - 10:SS
     $ = "1" + SS$: GOTO 10
9 SS$ = "0" + SS$
10 S$ =  STR$ (S%) + S$
11  REM ZEILE FUER DIAGNOSE
12  NEXT
13  PRINT " ": PRINT " ": PRINT "
     =============================="
     : PRINT "ERGEBNIS DER STRING
     ADDITION": PRINT "===========
     ===================": PRINT "
     "
14  PRINT "SUMMAND 1 (S1$)  : "S1$

15  PRINT "SUMMAND 2 (S2$)  : "S2$

16  PRINT "SUMME     (S$)   : "S$
17  PRINT " "
18  PRINT "=========================
     ======================"
19  END
]

]

==========================
ERGEBNIS DER STRINGADDITION
==========================

SUMMAND 1 (S1$)  : 112233445566778899
SUMMAND 2 (S2$)  : 192939495969798999
SUMME     (S$)   : 305172941536577898

====================================
```

Bild 3. Ein erster Vorversuch: Unverkürzte Addition über volle 18 Stellen Rechengenauigkeit

Stelle	MAX+1	MAX	••••	3	2	1	
STRING:							
S1 $:		1	••••	9	9	Ø	(Summand 1)
S2 $:		9	••••	1	1	Ø	(Summand 2)
SS $:	1	1	••••	1	Ø	Ø	(Übertrags-Kette)
S $:	1	1	••••	1	Ø	Ø	(Summen-Kette)

Bild 2. Dieses Schema für schriftliche Additionen, vielen Lesern aus der Grundschulzeit bestens bekannt, liegt auch der „String-Mathematik" zugrunde, mit deren Hilfe Computer exaktes Rechnen lernen

```
]
]
1  REM VERFAHRENSPRINZIP
2  REM ----------------
3 S1$ = "123"
4 S2$ = "192"
5 S$ = "":SS$ = "0":FIRST = 1
6  FOR N = 3 TO 1 STEP  - 1
7 S% =  VAL ( MID$ (S1$,N,1)) +  VAL
     ( MID$ (S2$,N,1)) +  VAL ( LEFT$
     (SS$,1))
8  IF S% > 9 THEN S% = S% - 10:SS
     $ = "1" + SS$: GOTO 10
9 SS$ = "0" + SS$
10 S$ =  STR$ (S%) + S$
11  GOSUB 21
12  NEXT
13  PRINT " ": PRINT " ": PRINT "
     ==========================="
     : PRINT "ERGEBNIS DER STRING
     ADDITION": PRINT "==========
     =================": PRINT "
     "
14  PRINT "SUMMAND 1 (S1$) : "S1$

15  PRINT "SUMMAND 2 (S2$) : "S2$

16  PRINT "SUMME      (S$)  : "S$
17  PRINT " "
18  PRINT "UEBERTRAG (SS$) : "SS$

19  PRINT "=====================
     ================="
20  END
21  IF FIRST THEN  PRINT " ": PRINT
     "DIAGNOSE:": PRINT "--------
     -": PRINT " ":FIRST = 0
22  PRINT " "
23  PRINT "STELLENNUMMER N : "N
24  PRINT "STELLENWERT  S% : "S%
25  PRINT "SUMMENSTRING S$ : "S$
26  PRINT " "
27  PRINT "UEBERTRAG (SS$) : "SS$

28  PRINT "---------------------
     ---------------": RETURN
```

Bild 4. Diagnose-Routinen zwingen den Rechner, zu zeigen, auf welche Weise er „String-Mathematik" betreibt

```
]
]

DIAGNOSE:
---------

STELLENNUMMER N : 3
STELLENWERT  S% : 5
SUMMENSTRING S$ : 5

UEBERTRAG (SS$) : 00
---------------------------------------

STELLENNUMMER N : 2
STELLENWERT  S% : 1
SUMMENSTRING S$ : 15

UEBERTRAG (SS$) : 100
---------------------------------------

STELLENNUMMER N : 1
STELLENWERT  S% : 3
SUMMENSTRING S$ : 315

UEBERTRAG (SS$) : 0100
---------------------------------------

===========================
ERGEBNIS DER STRINGADDITION
===========================

SUMMAND 1 (S1$) : 123
SUMMAND 2 (S2$) : 192
SUMME      (S$) : 315

UEBERTRAG (SS$) : 0100
=======================================
```

Bild 5. Die Druckausgabe verrät, wie sich der als Zeiger dienende Schleifenzähler beim Abtasten der Operandenstrings verändert und hierbei Ergebnis-String und Übertrags-String anwachsen

```
1  REM DEMONSTRATIONSPROGRAMM FUER ,STRINGMATHEMATIK'
2  REM =============================================

3  REM HANS-GEORG JOEPGEN, 9/80.

4  REM INITIALISIEREN
5  REM --------------
6  HOME : GOSUB 20: PRINT "  OPERATION MIT ERHOEHTER GENAUIGKEIT"
7  PRINT : PRINT "  ALS BEISPIEL FUER ,STRINGMATHEMATIK'"
8  PRINT : GOSUB 20: PRINT : PRINT : PRINT
9  PRINT " +++ DIAGNOSE GEWUENSCHT (J/N)? ";
10  GET A$: PRINT A$: IF (A$ <  > "J") AND (A$ <  > "N") THEN  INVERSE : PRINT
     CHR$ (7): PRINT "   NUR ,J' ODER ,N' BITTE!": NORMAL : PRINT : GOTO
     9
11 DIAGNOSE = (A$ = "J"): PRINT : PRINT
12  REM
13  REM HAUPTSCHLEIFE
14  REM -------------
15  GOSUB 20: INPUT " OPERAND 1:   ";S1$: PRINT : INPUT " OPERAND 2:   ";S2$
     : PRINT
16  GOSUB 24: PRINT : PRINT "* ERGEBNIS: "S$: PRINT : GOTO 15
17  REM
18  REM TRENNSTRICH
19  REM -----------
20  FOR N = 1 TO 40: PRINT "=";: NEXT : PRINT : RETURN
21  REM
22  REM STRINGADDITION
23  REM --------------
24 MAX =  LEN (S1$): IF  LEN (S2$) > MAX THEN MAX =  LEN (S2$)
25  IF  LEN (S1$) < MAX THEN S1$ = "0" + S1$: GOTO 25
26  IF  LEN (S2$) < MAX THEN S2$ = "0" + S2$: GOTO 26
27  PRINT : PRINT "= ADDITION ="
28 S$ = "":SS$ = "": IF DIAGNOSE THEN  GOSUB 42
29  FOR N = MAX TO 1 STEP  - 1
30 S% =  VAL ( MID$ (S1$,N,1)) +  VAL ( MID$ (S2$,N,1)) +  VAL ( LEFT$ (SS
     $,1))
31  IF S% > 9 THEN S% = S% - 10:SS$ = "1" + SS$: GOTO 33
32 SS$ = "0" + SS$
33 S$ =  STR$ (S%) + S$
34  IF DIAGNOSE THEN  GOSUB 46
35  NEXT
36  IF  LEFT$ (SS$,1) = "1" THEN S$ = "1" + S$: GOTO 38
37 S$ = " " + S$
38  RETURN
39  REM
40  REM DIAGNOSE EINS
41  REM -------------
42  PRINT " +++ DIAGNOSE:": PRINT "      ---------": PRINT : PRINT " S1$: "
     S1$: PRINT " S2$: "S2$: PRINT " MAX: "MAX: PRINT : RETURN
43  REM
44  REM DIAGNOSE ZWEI
45  REM -------------
46  PRINT : PRINT " STELLE N: "N: PRINT " S$:    "S$: PRINT " SS$:   "SS$: PRINT
     : RETURN
47  REM ------------------------
```

Bild 6. Ein voll praxistaugliches Betriebsprogramm für Additionen mit über 250 Stellen Genauigkeit, eingebettet in Test- und Diagnoserahmen

den eines solchen Strings von der Tastatur her und die unmittelbare Ausgabe des Strings auf Schirm oder Drucker erlauben, können wir beim Addieren großer Zahlen in Stringform nach „Fritzchens Schulheft-Algorithmus" verfahren, nach der Art, wie üblicherweise schriftlich addiert wird. Man schreibt (*Bild 2*) die beiden Summanden untereinander, läßt Raum für eine Übertrags-Zeile und sieht dann die Ergebnis-Zeile vor. In die äußerste rechte Stelle der Übertrags-Zeile kommt von vornherein eine Null, da es ja keine vorangegangene Rechnung und mithin auch keinen Übertrag aus dieser Rechnung gibt. Sodann addieren wir, mit der Stelle 1 beginnend und nach links fortschreitend, jeweils die Summanden und den Übertrag. Treten bei diesem Zusammenzählen selbst wieder Überträge auf, so werden sie in der Folgestelle notiert. *Bild 3* zeigt ein erstes kleines Basic-Programm und seinen Ergebnisausdruck: Die fehlerfreie Addition zweier achtzehnstelliger Zahlen ist auf Anhieb gelungen. Um deutlich zu machen, daß im Rechner justament das Gleiche geschieht wie in Fritzchens Rechenheft, erweitern wir das Programm, indem wir eine Diagnose-Ausgabe einfügen (*Bild 4*) – der Ausdruck (*Bild 5*) zeigt, wie sich der Summen-String und der Übertrags-String füllen.

Ein allgemeines Additionsverfahren

So überzeugend die Demonstration der grundsätzlichen Wirksamkeit von String-Mathematik bis hierhin auch ausgefallen sein mag, unser Vorführ-Beispiel krankt an zwei Mängeln, die es

unbrauchbar für die meisten praktischen Einsatzfälle machen: Beide Summanden mußten die gleiche Stellenzahl haben, und diese gleiche Stellenzahl steht als Programm-Konstante in Zeile 6 unverrückbar fest! Um unserer String-Addition zur Allgemeingültigkeit zu verhelfen, müssen wir diese Mängel ausmerzen – und wie das geschieht, zeigt ein neues Programm-Listing (*Bild 6*). In den Zeilen 6 bis 11 meldet sich das Programm auf dem Schirm und gewinnt eine Boole'sche Variable DIAGNOSE, die darüber entscheidet, ob später mit Hilfe der Subroutinen DIAGNOSE 1 und DIAGNOSE 2 die Stringveränderungen mitgeteilt werden. Zeile 24 lädt die Variable MAX mit der Stellenzahl des längeren der beiden Summanden-Strings. In den zwei folgenden Zeilen wird, sofern einer der beiden Summanden-Strings weniger als MAX Stellen hat, die betreffende Zeichenkette durch vorangestellte Nullen auf MAX Stellen aufgefüllt. Nun können wir uns erneut des Schulheft-Algorithmus bedienen. *Bild 7* zeigt, daß unsere Stringaddition damit jenen Grad an Freiraum zu meistern weiß, der ihr Praxis-Weihen verleiht.

```
========================================
  OPERATION MIT ERHOEHTER GENAUIGKEIT
  ALS BEISPIEL FUER ,STRINGMATHEMATIK'
========================================
 +++ DIAGNOSE GEWUENSCHT (J/N)? N
========================================
 OPERAND 1:  1112223334445556667778889999
 OPERAND 2:  112233445566778899
= SUBTRAKTION =
* ERGEBNIS   1112223333323222221211110100
========================================
 OPERAND 1:  123456789
 OPERAND 2:  9988776655443322110000000000
= SUBTRAKTION =
* ERGEBNIS  -9988776655443322108765432111
========================================
```

Bild 7. Die Fesseln sind gefallen: Keine Beschränkungen für den Definitionsbereich des Algorithmus mehr

Subtraktion – nicht weniger universell

Kommen wir zur zweiten Grundrechenart, der Subtraktion. Hier können wir uns weitgehend des gleichen Lösungsweges bedienen, der auch bei String-Addition weiterhalf. Es sind freilich einige Änderungen vorzunehmen. S1$ steht hier für Minuend, S2$ für Subtrahend, S$ ist die gewonnene Differenz. Bei der Stellenbearbeitung und der Übertrags-Behandlung folgen wir nun den Subtraktionsgesetzen – und: Eine weitere Boole'sche Variable kommt ins Spiel, sie heißt MINUS und zeigt an, daß ein negatives Ergebnis eintreten wird. Ist MINUS wahr, dann müssen wir Subtrahend und Minuend vertauschen und unseren Ergebnis-String statt mit einer führenden Leerstelle (SPACE) mit einem Minus-Zeichen versehen. Die komplette Subroutine Stringsubtraktion findet sich in *Bild 8*, einen Probelauf mit hinwiederum überzeugendem Ergebnis bringt *Bild 9*. An dieser Stelle dient es sicherlich der Vertiefung, noch einmal mit Hilfe unserer Diagnose gezielt zu verfolgen, was geschieht, wenn unser Computer Strings subtrahiert – um Raum zu sparen, nehmen wir diesmal Operanden mit geringer Stellenzahl, die man in der Praxis sicherlich „stringfrei" programmiert (Bild 10). Da Operand 1 kleiner ist als Operand 2, wurde die Boole'sche Variable MINUS auf „Wahr" gesetzt. Wir erkennen dies daraus, daß die Inhalte von S1$ und S$ zu Diagnose-Beginn bereits getauscht erscheinen. MAX ist richtig auf drei gesetzt, der Rechner beginnt „rechtsaußen" bei N=MAX. Erstes Ergebnis 7, kein Übertrag – ebenfalls kein Überlauf bei Behandlung der Stellen 2 und 1. Im Ergebnis-Druck erscheint ein Minus-Zeichen, da MINUS gesetzt war.

```
21  REM
22  REM STRINGSUBTRAKTION
23  REM -----------------
24 MAX =  LEN (S1$): IF  LEN (S2$) > MAX THEN MAX =  LEN (S2$)
25  IF  LEN (S1$) < MAX THEN S1$ = "0" + S1$: GOTO 25
26  IF  LEN (S2$) < MAX THEN S2$ = "0" + S2$: GOTO 26
27  PRINT : PRINT "= SUBTRAKTION =":MINUSFLAG = 0: IF S1$ < S2$ THEN SS$ =
     S1$:S1$ = S2$:S2$ = SS$:MINUSFLAG = 1
28 S$ = "":SS$ = "": IF DIAGNOSE THEN  GOSUB 42
29  FOR N = MAX TO 1 STEP  - 1
30 S% =  VAL ( MID$ (S1$,N,1)) -  VAL ( MID$ (S2$,N,1)) -  VAL ( LEFT$ (SS
     $,1))
31  IF S% < 0 THEN S% = S% + 10:SS$ = "1" + SS$: GOTO 33
32 SS$ = "0" + SS$
33 S$ =  STR$ (S%) + S$
34  IF DIAGNOSE THEN  GOSUB 46
35  NEXT : IF MINUSFLAG THEN S$ = "-" + S$: GOTO 38
36  IF  LEFT$ (SS$,1) = "1" THEN S$ = "1" + S$: GOTO 38
37 S$ = " " + S$
38  RETURN
39  REM
```

Bild 8. Dieses Unterprogramm, bestimmt für das Rahmenprogramm aus Bild 6, handhabt mühelos Stringsubtraktionen beliebiger Genauigkeit

Höhere Rechenarten: Ebenfalls „string-kompatibel"

Für den geforderten Entwicklungszweck, zu dessen Realisierung „Stringmathematik" geschaffen wurde, waren Subtraktion und Addition gefordert, doch lassen sich nach den hier gezeigten Algorithmen unschwer auch höhere Rechenarten in Zeichenketten-Technik verwirklichen: Bei der Multiplikation wird man, ebenfalls in Schleife, zeichenweise multiplizieren und die so gewonnenen Teilprodukte schließlich stellenrichtig addieren – unter Benutzung von STRING-ADDITION ein Leichtes – und sinngemäßes gilt für die Division: Es geht allein darum, die vom schriftlichen Rechnen her gewohnten Lösungswege wohlstrukturiert in gleicher Weise umzusetzen, wie das für die ersten beiden Grundrechenarten vorgeführt wurde. Übrigens: Auch für schriftliches Wurzelziehen existiert ein Algorithmus, der sich unschwer in String-Operationen umsetzen läßt – allerdings, Fälle, in denen Wurzeln höherer Genauigkeit verlangt werden, dürften wohl äußerst selten sein.

Dezimalbrüche – kein Problem für „Stringmathematik"

Eher von praktischem Wert ist eine Erweiterung der String-Mathematik auf den Bereich der Dezimalzahlen mit Nachkomma-Stellen. Auch dies ist möglich und bedarf nur zweier Erweiterungen: Die Operanden-Strings müssen hier, wie üblich, auf gleiche Vorkomma-Stellenzahl und danach noch zusätzlich auf gleiche Nachkomma-Stellenzahl gebracht werden; auch hier durch Auffüllen mit Nullen. Sodann ist in die eigentliche Arbeitsschleife eine Prüfung auf Komma einzufügen und beim Vorliegen dieser Bedingung ein Sprung zum korrespondierenden NEXT-Statement zu programmieren. Vorgeschlagener Weg:
IF MID$(S1$,N,1) = CHR$ (44) GOTO ...
Die gleichlautend formulierte Prüfung verrichtet auch beim anfänglichen komma-bündigen Formatieren beider Operandenstrings gute Dienste. Des weiteren wird man bei der Benutzung von String-Mathematik gut daran tun, solide Ordnung bei der String-Übergabe zwischen dem Hauptprogramm und diversen Subroutinen zu halten und dabei feste Konventionen zu beachten, wie etwa die, daß die Bezeichnung der Operanden dem Muster S1$, S2$, S3$... folgt und Hauptergebnisse in S$, Nebenergebnisse in SS$ zurückkommen.

Grenzen und Verbesserungsmöglichkeiten

Die Anzahl der Stellen, die Stringmathematik zu handhaben weiß, wird allein durch den verwendeten Interpreter oder Compiler begrenzt, doch erlauben selbst frühe Versionen von Anfängermaschinen wie die ersten PETs Ketten mit 255 Zeichen. Hier setzt nicht die Firmware Schranken, sondern bestenfalls die Rechengeschwindigkeit, die annähernd linear mit der Länge der Kette sinkt.

```
========================================
  OPERATION MIT ERHOEHTER GENAUIGKEIT
  ALS BEISPIEL FUER 'STRINGMATHEMATIK'
========================================
 +++ DIAGNOSE GEWUENSCHT (J/N)? N
========================================
 OPERAND 1:  11022033044055066077088099 0
   ERAND 2:  990880770660550440330220110
   DITION =
*  ERGEBNIS: 1101101101101101101101101100
========================================
 OPERAND 1:  112321233123412351236123712
 OPERAND 2:  987654321
= ADDITION =
* ERGEBNIS:  112321233123412352223778033
========================================
```

Bild 9. In weniger als einer Sekunde gelöst: Die Aufgabe, eine achtzehnstellige Zahl von einer Zahl mit 27 Stellen zu subtrahieren

```
========================================
  OPERATION MIT ERHOEHTER GENAUIGKEIT
  ALS BEISPIEL FUER 'STRINGMATHEMATIK'
========================================
 +++ DIAGNOSE GEWUENSCHT (J/N)? J
========================================
 OPERAND 1:  12
 OPERAND 2:  999
= SUBTRAKTION =
 +++ DIAGNOSE:
     ----------
 S1$: 999
 S2$: 012
   X: 3
  ELLE N: 3
 S$:   7
 SS$:  0
 STELLE N: 2
 S$:   87
 SS$:  00
 STELLE N: 1
 S$:   987
 SS$:  000
* ERGEBNIS: -987
========================================
```

Bild 10. Die Meldungen der Überwachungsroutine DIAGNOSE zeigen, wie der Rechner bei der String-Subtraktion Schritt um Schritt, Stelle um Stelle vorgeht

Stringverarbeitung kostet Rechenzeit, freilich weniger, als erwartet: Für die (etwas lebensferne) Subtraktion zweier Zahlen mit je hundert Stellen nach dem Programm aus Bild 6/Bild 8 benötigte ein ITT-2020 etwa 5 Sekunden – die Zeit zur Ergebnisausgabe schon inbegriffen. Eine gewisse Schwäche des vorgestellten Verfahrens soll nicht verschwiegen werden: Mehr-Operanden-Rechnungen, bei denen Überläufe mit Werten größer Eins vorkommen, sind mit dem beschriebenen Lösungsweg nicht zu meistern, dies würde gründliche Änderungen des Rechenablaufs erforderlich machen. So wird man denn gegebenfalls Listenadditionen und ähnliche Kettenrechnungen vom Programm her in Zwei-Operanden-Operationen aufzulösen haben.



Literatur

[1] *Joepgen, Hans-Georg*: Vorsicht – Falle! Die „Null-Probleme" binärer Basic-Interpreter. FUNKSCHAU 1980, Heft 2, S. 79. Franzis-Verlag, München.

[2] *Joepgen, Hans-Georg*: Gute Noten dank PET – Rechenfehler eines Hobbycomputers. FUNKSCHAU 1979, Heft 9, S. 525. Franzis-Verlag, München.

[3] *Joepgen, Hans-Georg*: Kommissar deckt Rechenungenauigkeit auf. Sonderheft „Hobbycomputer 1". Franzis-Verlag, München.

[4] *Handle, Dr. Franz*: Zahlendarstellung im PET. Sonderheft „Hobbycomputer 2", Franzis-Verlag, München.

[5] *Bowles, Professor Kenneth L. und andere*: Apple Pascal Reference Manual. Apple Computer Incorporated. Cupertino, Kalifornien.

[6] *Kaucher, Dr. Edgar; Klatte, Dr. Rudi und Ullrich; Dr. Christian*: Höhere Programmiersprachen. B.I. Wissenschaftsverlag, Mannheim.

[7] *Willis, Jerry und Pol, Bernd*: Was der Mikrocomputer alles kann. Vogel-Verlag, Würzburg.

[8] *Wirth, Niklaus*: Algorithms plus Data Structures = Programs. Prentice Hall, Englewood Cliffs, New Jersey.

[9] *Lübbert, William F.*: What's where in the Apple. MICRO Nr. 15. Europa-Vertrieb: Computershop, Markdorf.

[10] *Joepgen, Hans-Georg*: Bit-Flags ermöglichen elegante Basic-Wege. Sonderheft „Programme". Franzis-Verlag, München.

[11] *Joepgen, Hans-Georg*: Der Euro-Apple. FUNKSCHAU 1979, Heft 14, S. 831...835.

[12] *Joepgen, Hans-Georg*: Variablen-Wächter sorgt für mehr Programm-Transparenz. ELEKTRONIK 1980, Heft 8, S. 87...88.

[13] *Joepgen, Hans-Georg*: Basic-Programm simuliert Wobbel-Meßplatz. ELEKTRONIK 1980, Heft 2, S. 49...55.

[14] Handbücher der Hersteller zu folgenden Rechnern: Apple II, BASF 7100, CBM, TRS-80, PC-100, ITT 2020, DAI, TI 99/4, Microset 8080, CS 2000 und SDC-85.

Die verwendeten String-Operatoren

A$ = CHR$ (x)	Der Zeichenkette A$ wird das Zeichen zugewiesen, dessen ASCII-Code durch die dezimale Zahl x beschrieben ist.
A$ = "123"	Die Zeichenkette A$ wird mit den ASCII-Zeichen der Ziffern 1, 2 und 3 geladen.
A$ = ""	Die Zeichenkette A$ wird gelöscht („Leerstring").
S% = VAL (A$)	Der Integer-Variablen S% wird jener Wert zugeordnet, den die ASCII-Zeichen in A$ beschreiben.
S$ = STR$ (S%)	Aus der Zahl in der Variablen S% wird das ASCII-Zeichen oder werden die ASCII-Zeichen gebildet, die diese Zahl beschreiben.
SS$ = "0" + SS$	Am Beginn der Kette SS$ wird das ASCII-Zeichen für "0" eingefügt.
A$ = RIGHT$ (B$,3)	A$ wird mit den letzten drei Zeichen der Kette B$ geladen.
A$ = LEFT$ (B$,5)	A$ wird mit den ersten fünf Zeichen von B$ geladen.
A$ = MID$ (B$,N,1)	A$ wird mit einem Zeichen aus B$ geladen – nämlich mit dem, das an der Stelle N in der Kette steht.
A = LEN (A$)	Der Variablen A wird als neuer Wert die Anzahl der Zeichen in der Kette A$ zugewiesen.

Die verwendeten String-Operatoren sind u. a. kompatibel mit den auf Maschinen folgender Firmen verwendeten Basic-Dialekten: Apple, BASF, Commodore, Siemens, Standard-Elektrik Lorenz, Tandy u. a. Teilkompatibilität: Texas Instruments u. a. Nicht kompatibel z. B.: Hewlett-Packard, General Electric Mark II und III, ältere Siemens- und Telefunken-Dialekte.